15-3 Stacks

A stack is a linear list in which all additions and deletions are restricted to one end, called the top. Stacks are known as the last in–first out (LIFO) data structure.

Topics discussed in this section:

Stack Structures
Stack Algorithms
Stack Demonstration

Note

A stack is a last in–first out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.

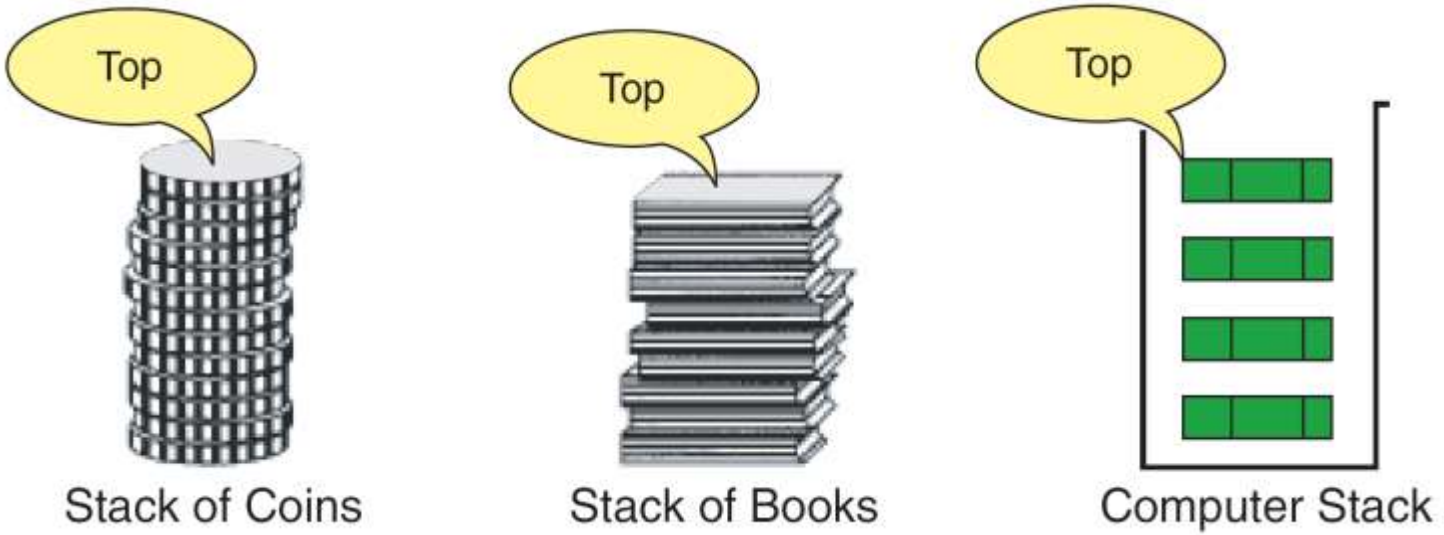


FIGURE 15-17 Stack

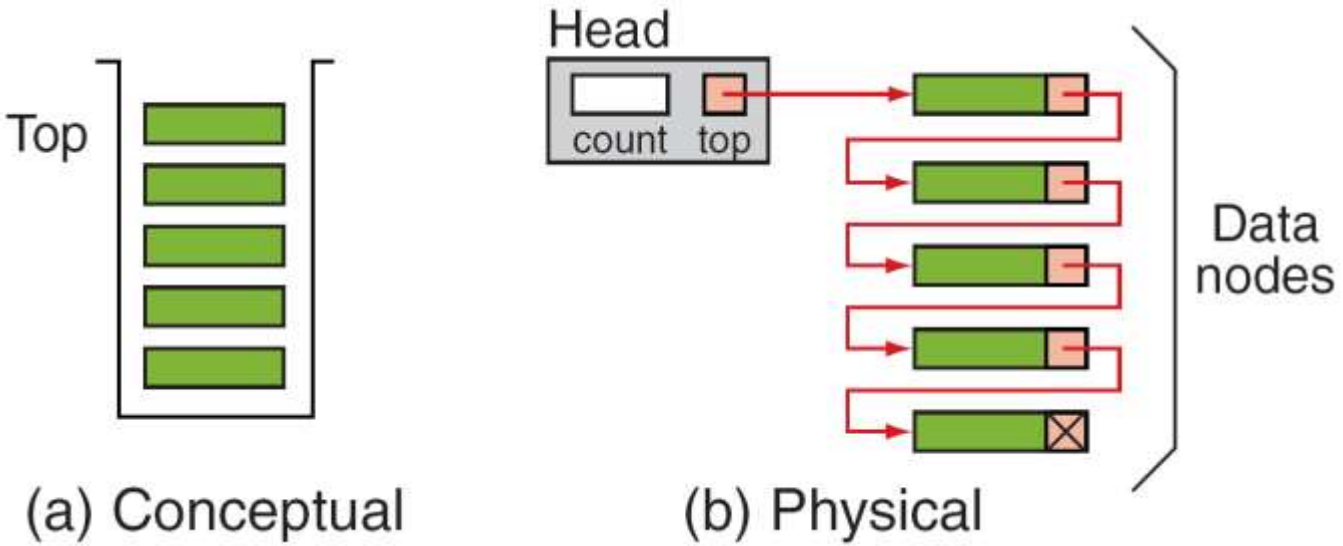
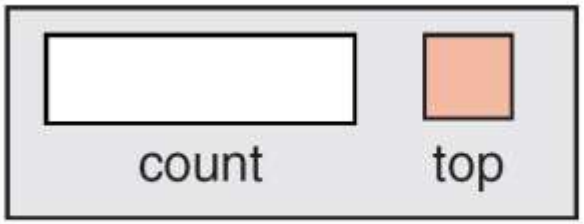
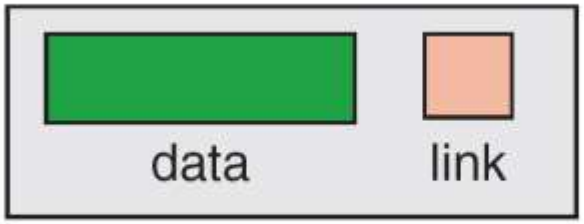


FIGURE 15-18 Conceptual and Physical Stack Implementations



Stack Head Structure



Stack Node Structure

```
typedef struct
   {
    int           count;
    struct node* top;
   } STACK;
typedef struct node
   {
    int           data;
    struct node* link;
   } STACK_NODE;
```

FIGURE 15-19 Stack Data Structure

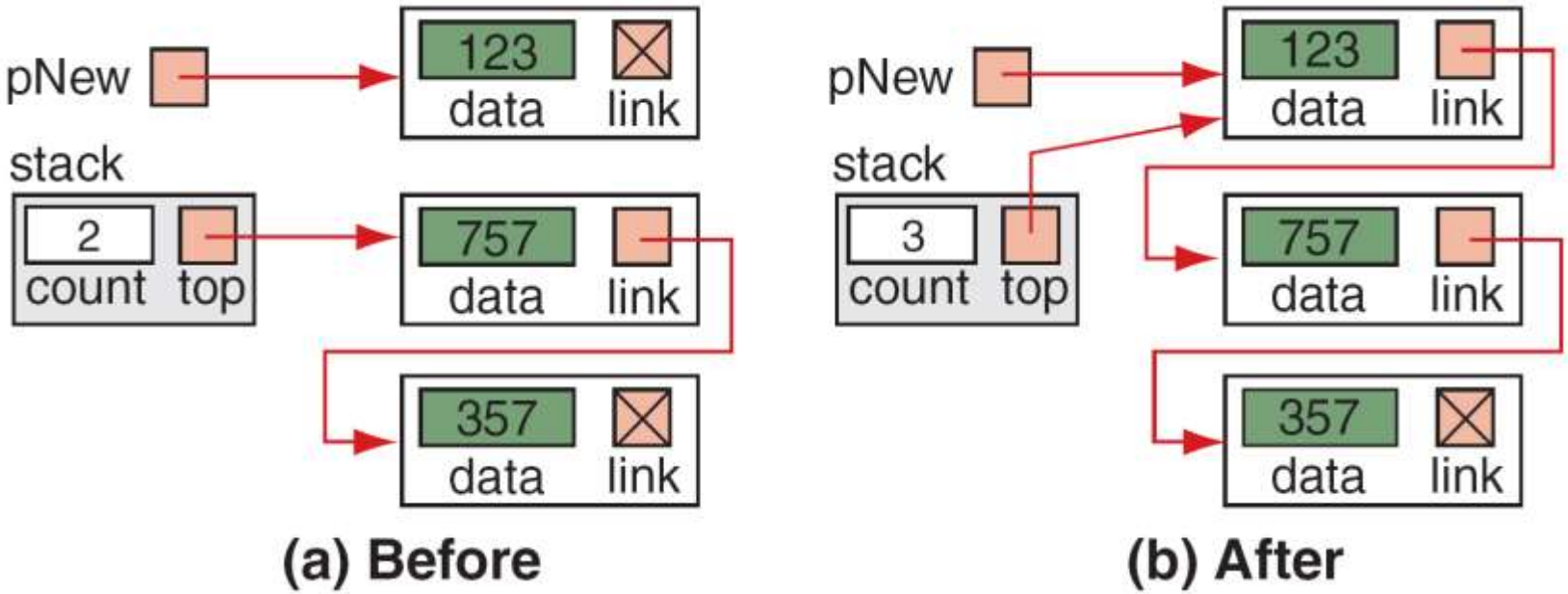


FIGURE 15-20 Streams

PROGRAM 15-9 Push Stack

```
1  /* ==================== push ====================
2     Inserts node into linked list stack.
3        Pre    pStack is pointer to valid stack header
4        Post   dataIn inserted
5        Return true  if successful
6               false if overflow
7  */
8  bool push (STACK* pStack, int dataIn)
9  {
10 // Local Declarations
11    STACK_NODE* pNew;
12    bool        success;
13
14 // Statements
15    pNew = (STACK_NODE*)malloc(sizeof (STACK_NODE));
16    if (!pNew)
17        success = false;
```

PROGRAM 15-9 Push Stack

```
18      else
19          {
20           pNew->data  = dataIn;
21           pNew->link  = pStack->top;
22           pStack->top = pNew;
23           pStack->count++;
24           success = true;
25          } // else
26      return success;
27  }   // push
```

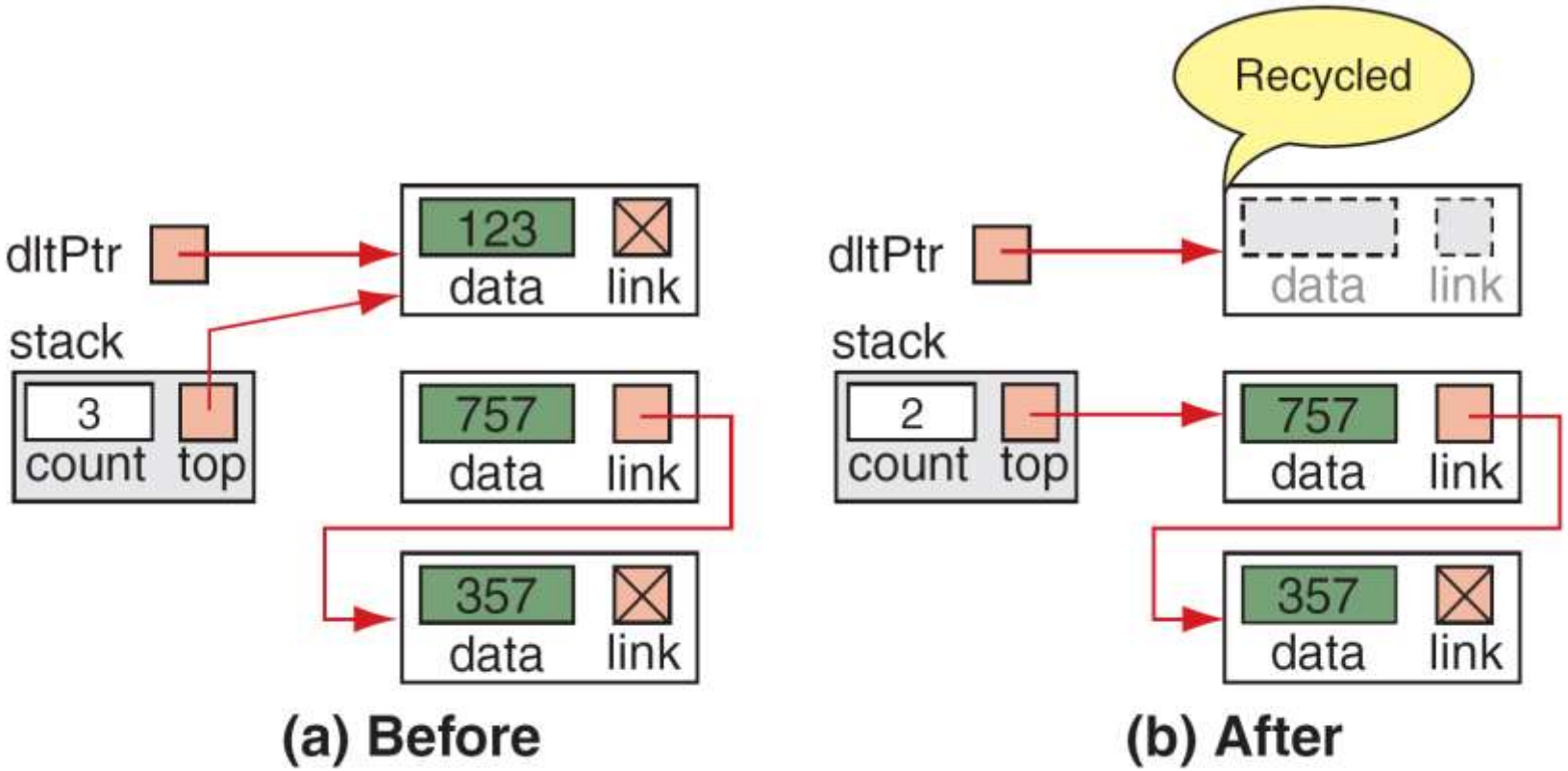


FIGURE 15-21 **Pop Stack Example**

PROGRAM 15-10 Pop Stack

```
 1  /* ==================== pop ====================
 2     Delete node from linked list stack.
 3        Pre     pStackTop is pointer to valid stack
 4        Post    dataOut contains deleted data
 5        Return  true  if successful
 6                false if underflow
 7  */
 8  bool pop (STACK* pStack, int* dataOut)
 9  {
10  // Local Declarations
11     STACK_NODE* pDlt;
12     bool        success;
13
14  // Statements
15     if (pStack->top)              // Test for Empty Stack
16        {
17         success  = true;
18         *dataOut = pStack->top->data;
```

PROGRAM 15-10 Pop Stack

```
20          pStack->top = (pStack->top)->link;
21          pStack->count--;
22          free (pDlt);
23        } // else
24     else
25        success = false;
26     return success;
27  }  // pop
```

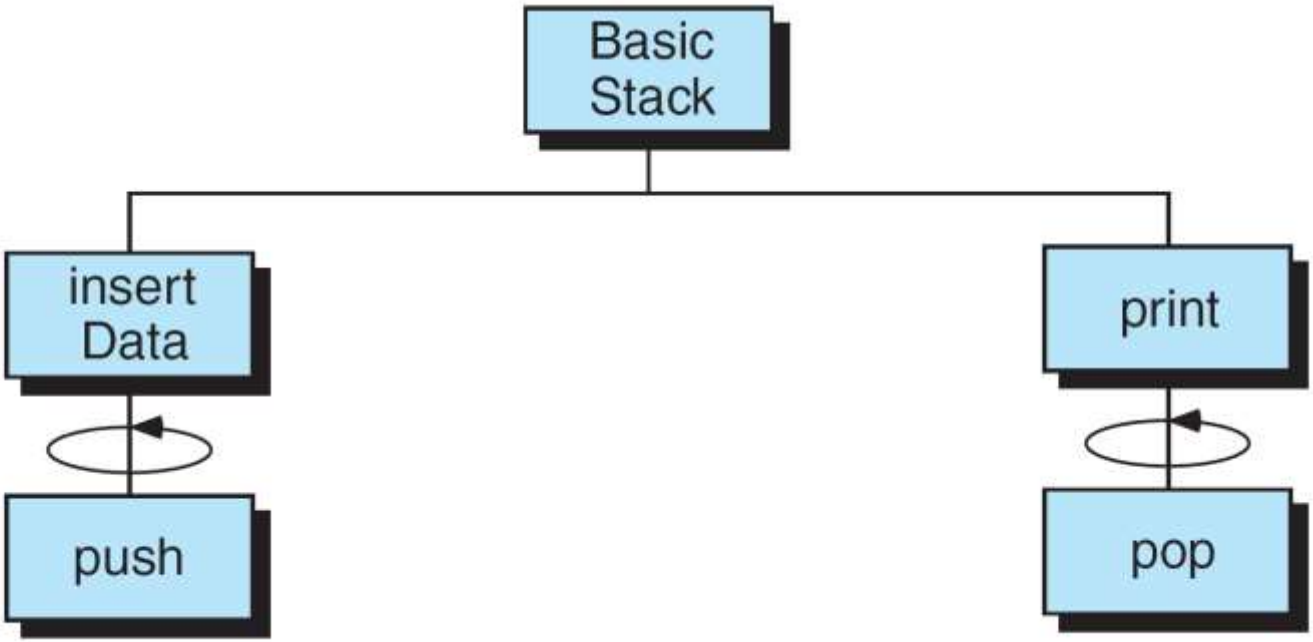


FIGURE 15-22 **Design for Basic Stack Program**

PROGRAM 15-11 Simple Stack Application Program

```
1  /* This program is a test driver to demonstrate the
2     basic operation of the stack push and pop functions.
3        Written by:
4        Date:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <stdbool.h>
9
10 // Global Declarations
11 typedef struct node
12    {
13     int          data;
14     struct node* link;
15    } STACK_NODE;
16
17 typedef struct
```

PROGRAM 15-11 Simple Stack Application Program

```
18      {
19       int          count;
20       STACK_NODE* top;
21      } STACK;
22
23  // Function Declarations
24  void insertData (STACK* pStack);
25  void print      (STACK* pStack);
26  bool push       (STACK* pList, int  dataIn);
27  bool pop        (STACK* pList, int* dataOut);
28
29  int main (void)
30  {
31  // Local Declarations
32  STACK* pStack;
33
34  // Statements
35     printf("Beginning Simple Stack Program\n");
36
```

PROGRAM 15-11 Simple Stack Application Program

```
37      pStack = malloc(sizeof(STACK));
38      if (!pStack)
39         printf("Error allocating stack"), exit(100);
40
41      pStack->top   = NULL;
42      pStack->count = 0;
43      insertData (pStack);
44      print      (pStack);
45
46      printf("\nEnd Simple Stack Program\n");
47      return 0;
48  }  // main
```

```
Results:
Beginning Simple Stack Program
Creating numbers:  854 763 123 532  82 632  33 426 228  90
Stack contained:    90 228 426  33 632  82 532 123 763 854
End Simple Stack Program
```

PROGRAM 15-12 Insert Data

```
/* ================= insertData =================
   This program creates random numbers and
   inserts them into a linked list stack.
      Pre  pStack is a pointer to first node
      Post Stack has been created
*/
void insertData (STACK* pStack)
{
// Local Declarations
   int  numIn;
   bool success;

// Statements
   printf("Creating numbers: ");
   for (int nodeCount = 0; nodeCount < 10; nodeCount++)
       {
        // Generate random number
        numIn   = rand() % 999;
        printf("%4d", numIn);
        success = push(pStack, numIn);
```

PROGRAM 15-12 Insert Data

```
21            if (!success)
22               {
23                printf("Error 101: Out of Memory\n");
24                exit (101);
25               } // if
26            } // for
27      printf("\n");
28      return;
29   }  // insertData
```

PROGRAM 15-13 Print Stack

```
 1  /* ====================== print ====================
 2     This function prints a singly linked stack.
 3        Pre     pStack is pointer to valid stack
 4        Post    data in stack printed
 5  */
 6  void print (STACK* pStack)
 7  {
 8  // Local Declarations
 9     int printData;
10
11  // Statements
12     printf("Stack contained:  ");
13     while (pop(pStack, &printData))
14         printf("%4d", printData);
15     return;
16  }  // print
```

15-4 Queues

A queue is a linear list in which data can be inserted only at one end, called the rear, and deleted from the other end, called the front.

Topics discussed in this section:

Queue Operations
Queue Linked List Design
Queue Functions
Queue Demonstration

Note

A queue is a linear list in which data can be inserted at one end, called the rear, and deleted from the other end, called the front. It is a first in–first out (FIFO) restricted data structure.

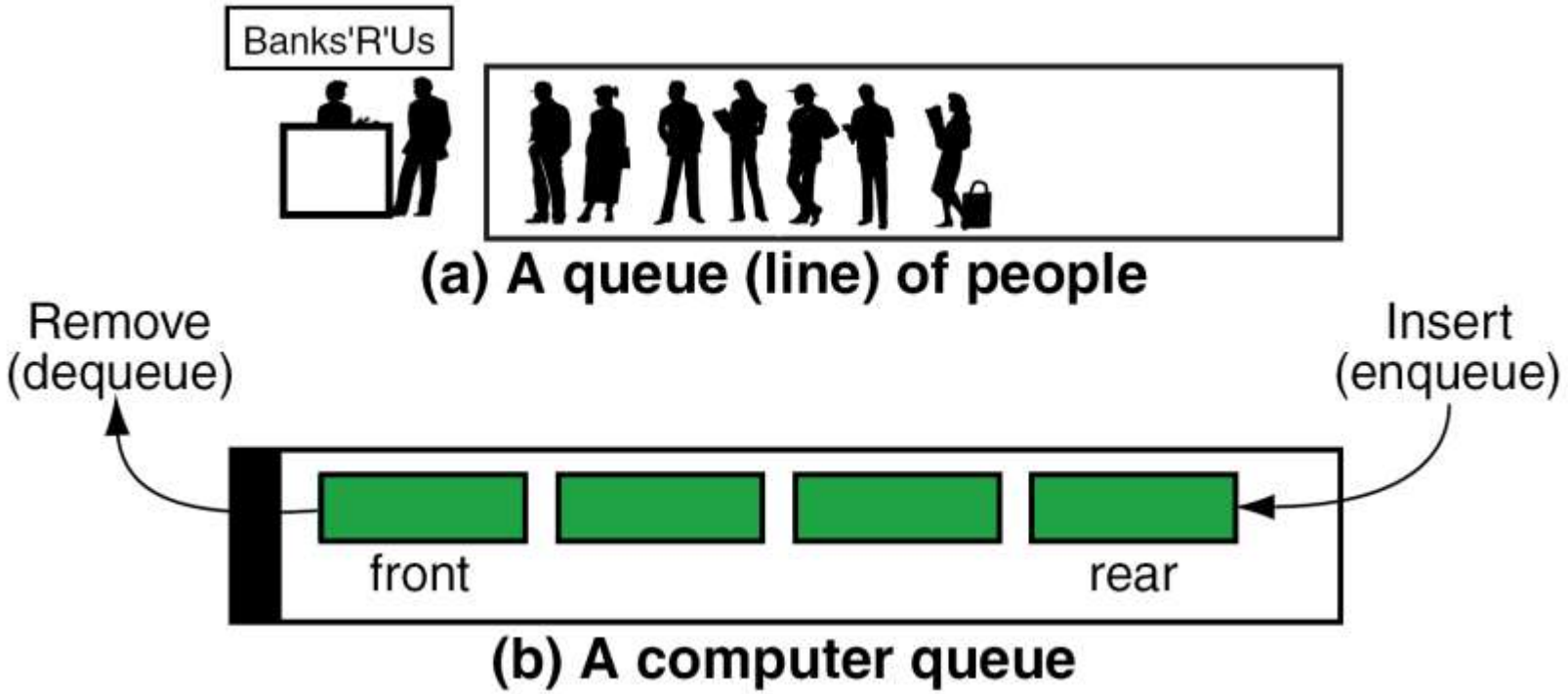


FIGURE 15-23 Queue Concept

Note

Enqueue inserts an element at the rear of the queue.

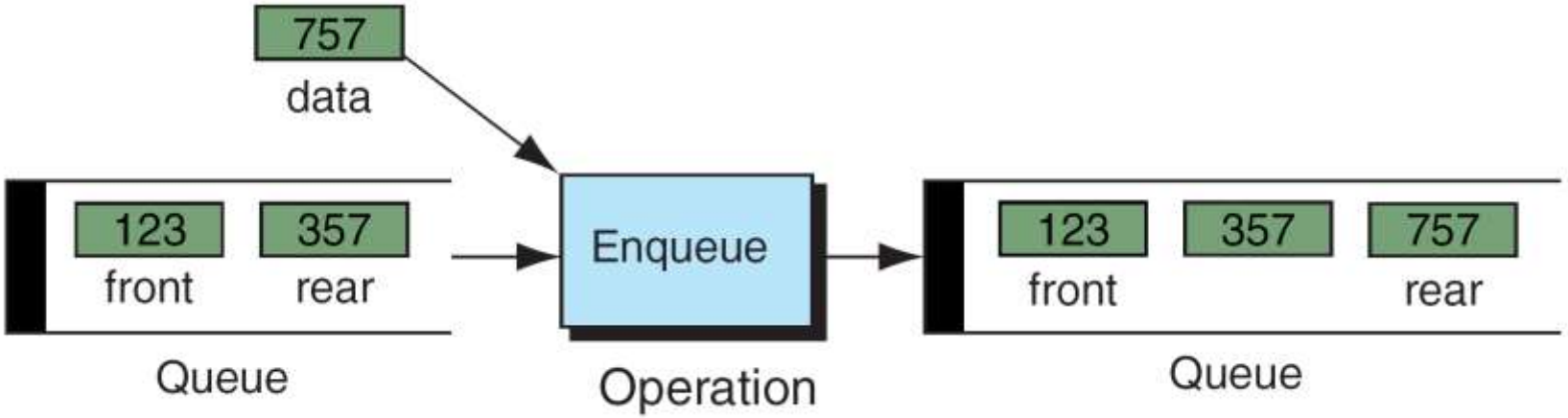


FIGURE 15-24 Enqueue

Note

Dequeue deletes an element at the front of the queue.

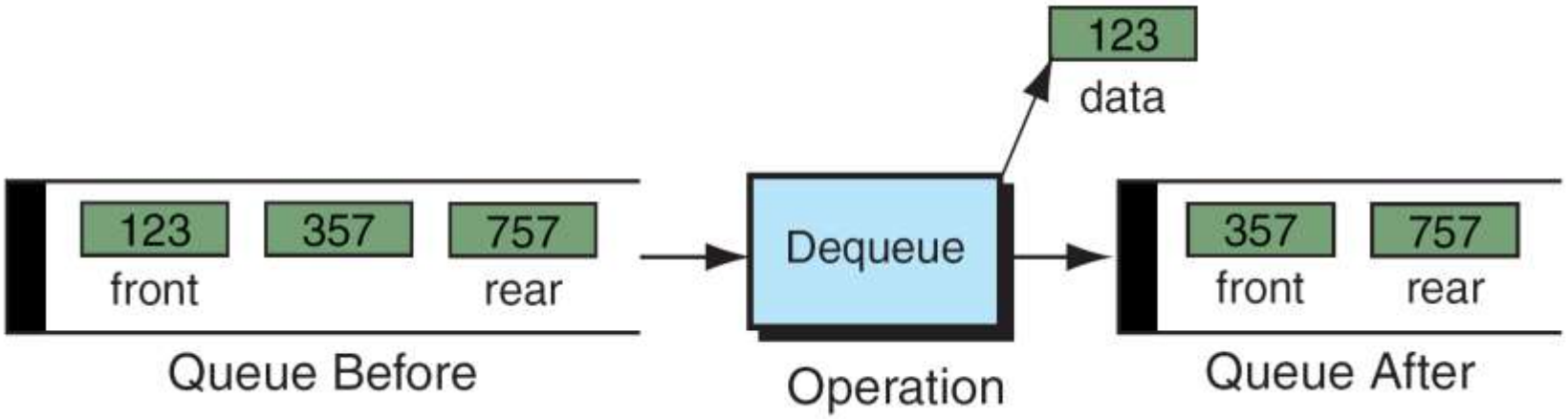


FIGURE 15-25 Dequeue

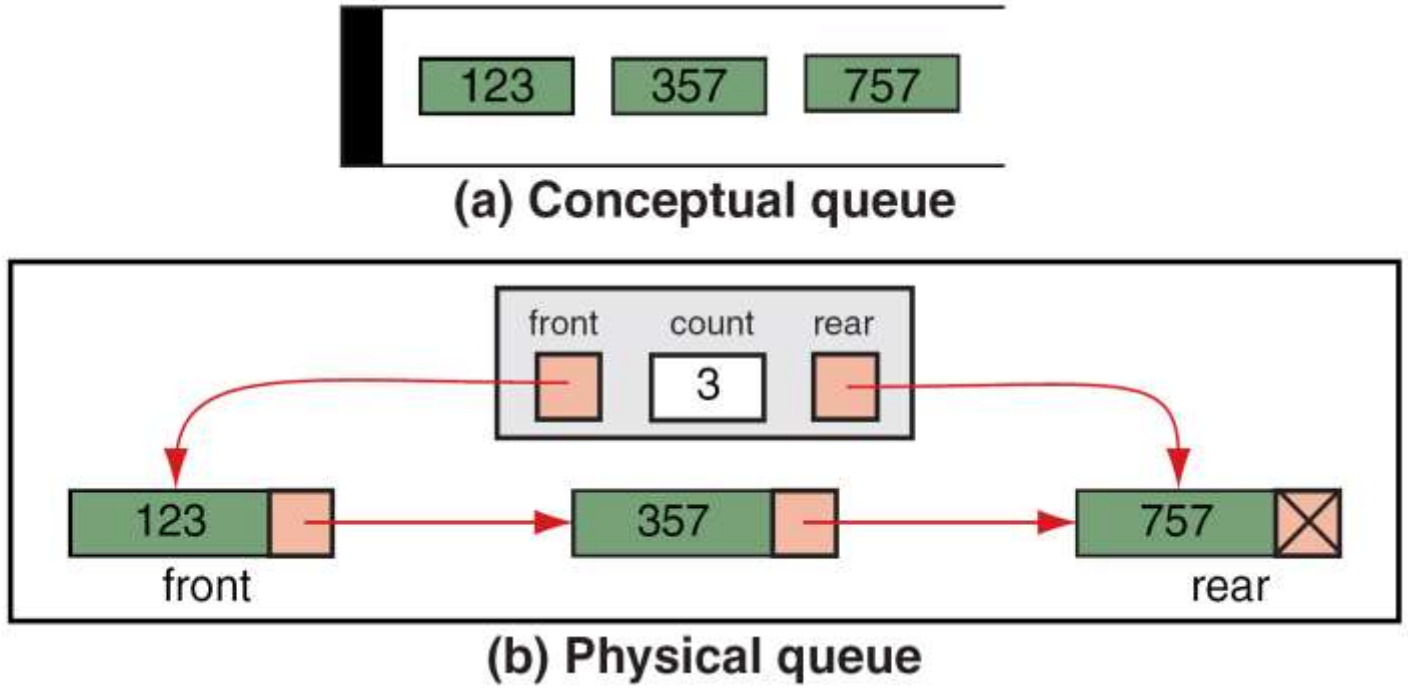
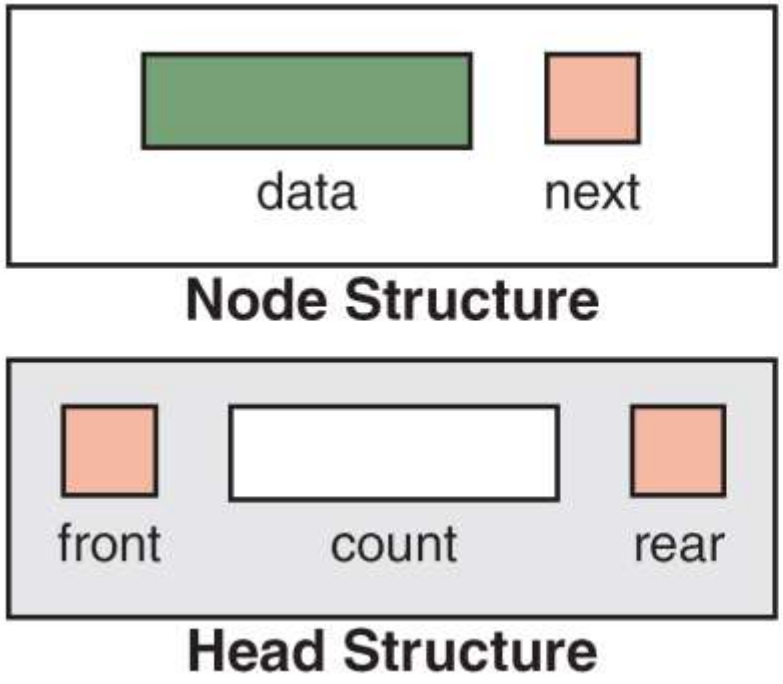


FIGURE 15-26 **Conceptual and Physical Queue Implementations**




```
typedef struct node
    {
     int            data;
     struct node*   next;
    } QUEUE_NODE;

typedef struct
    {
     QUEUE_NODE*  front;
     int          count;
     QUEUE_NODE*  rear;
    } QUEUE;
```

FIGURE 15-27 Queue Data Structure

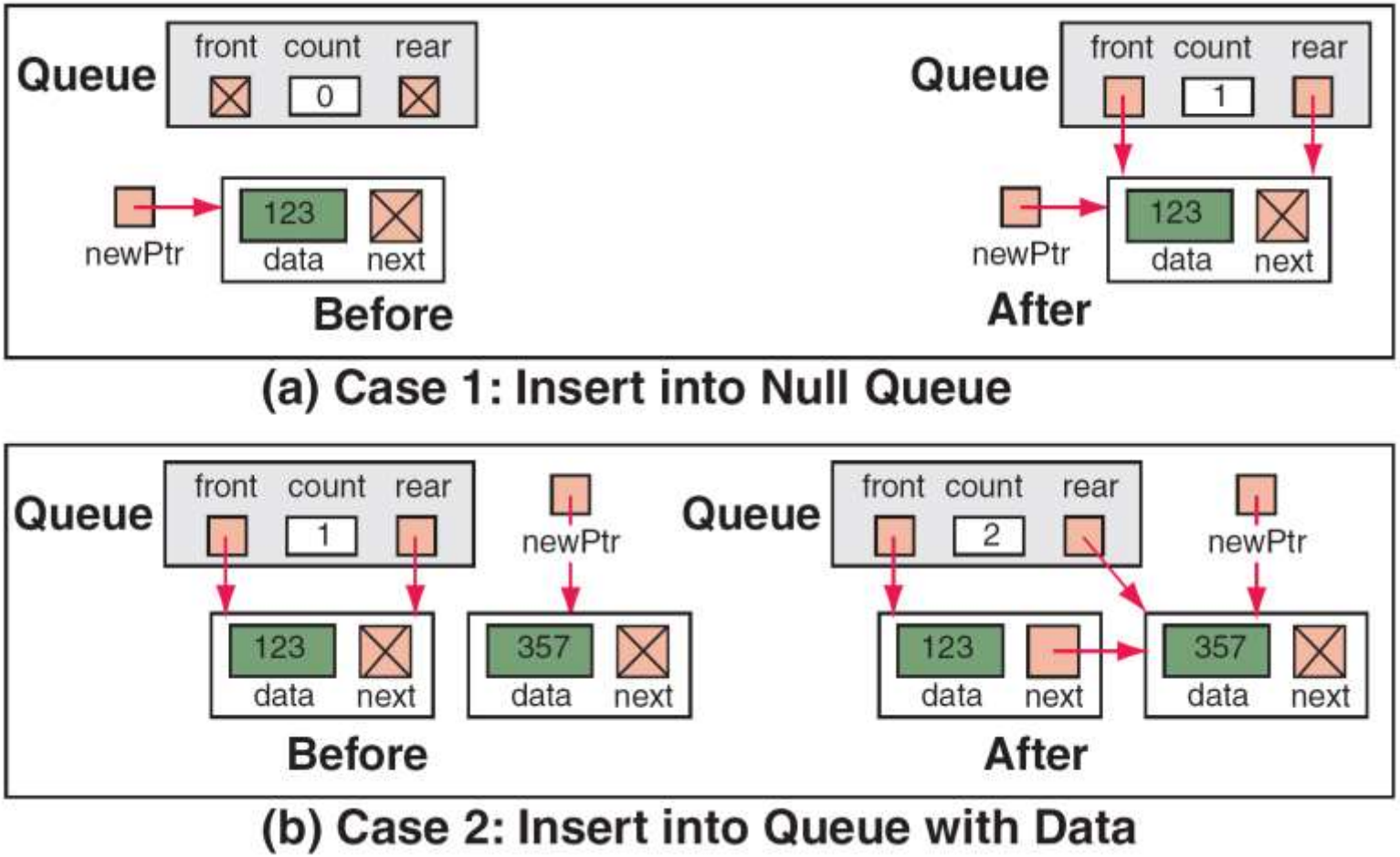


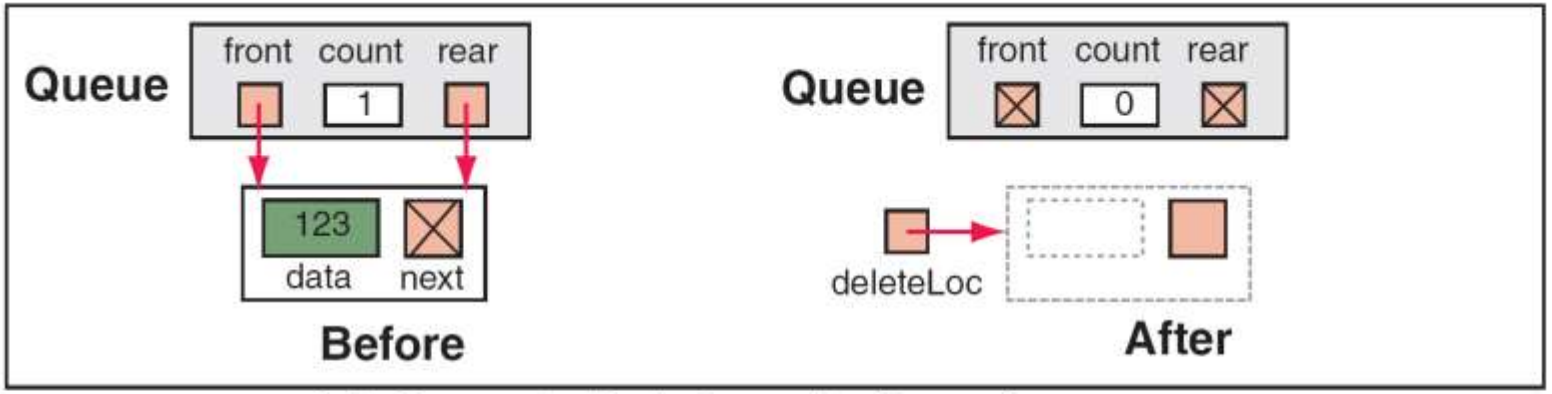
FIGURE 15-28 Enqueue Example

PROGRAM 15-14 Enqueue

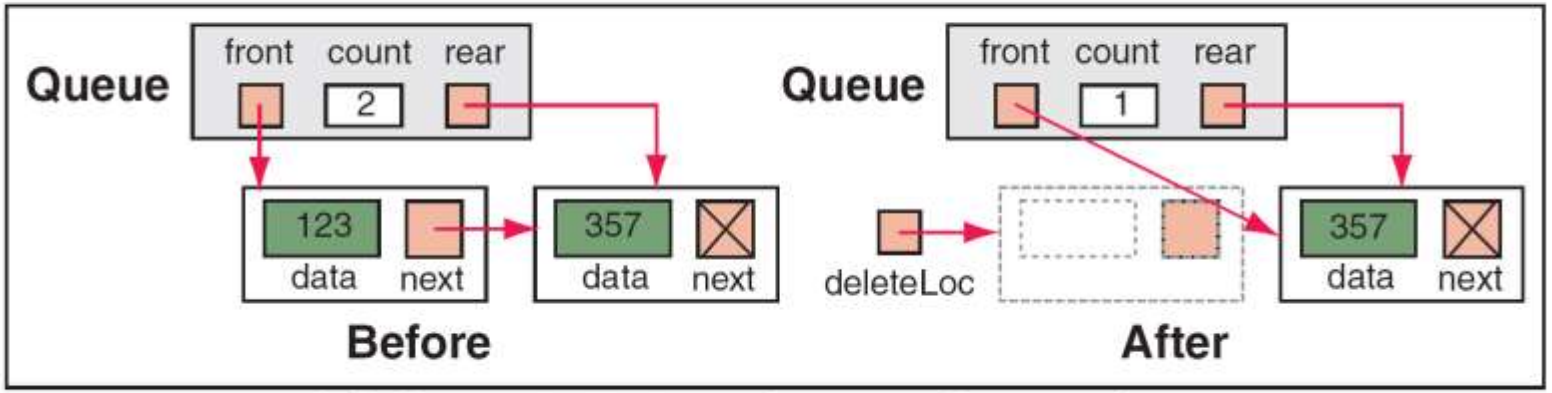
```
1  /* ================ enqueue ================
2     This algorithm inserts data into a queue.
3        Pre    queue is valid
4        Post   data have been inserted
5        Return true if successful, false if overflow
6  */
7  bool enqueue (QUEUE* queue, int dataIn)
8  {
9  // Local Declarations
10    QUEUE_NODE* newPtr;
11
12 // Statements
13    if (!(newPtr = malloc(sizeof(QUEUE_NODE))))
14       return false;
15
16    newPtr->data = dataIn;
17    newPtr->next = NULL;
18
```

PROGRAM 15-14 Enqueue

```
19      if (queue->count == 0)
20         // Inserting into null queue
21         queue->front  = newPtr;
22      else
23         queue->rear->next = newPtr;
24      (queue->count)++;
25      queue->rear = newPtr;
26      return true;
27   }  // enqueue
```



(a) Case 1: Delete only item in queue



(b) Case 2: Delete item at front of queue

FIGURE 15-29 Dequeue Examples

PROGRAM 15-15 Dequeue

```
/* ================= dequeue ================
   This algorithm deletes a node from the queue.
      Pre    queue is pointer to queue head structure
             dataOut is pointer to data being deleted
      Post   Data pointer to queue front returned and
             front element deleted and recycled.
      Return true if successful; false if underflow
*/
bool dequeue (QUEUE* queue, int* dataOut)
{
// Local Declarations
   QUEUE_NODE* deleteLoc;

// Statements
   if (!queue->count)
       return false;

```

PROGRAM 15-15 Dequeue

```
18      *dataOut = queue->front->data;
19      deleteLoc = queue->front;
20      if (queue->count == 1)
21          // Deleting only item in queue
22          queue->rear  = queue->front = NULL;
23      else
24          queue->front = queue->front->next;
25      (queue->count)--;
26      free (deleteLoc);
27
28      return true;
29  }   // dequeue
```

PROGRAM 15-16 Simple Queue Demonstration

```
 1  /* This program is a test driver to demonstrate the
 2     basic operation of the enqueue and dequeue functions.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include <string.h>
 9  #include <stdbool.h>
10
11  // Global Declarations
12  typedef struct node
13     {
14      int           data;
15      struct node* next;
16     } QUEUE_NODE;
17
18  typedef struct
19     {
20      QUEUE_NODE* front;
```

PROGRAM 15-16 Simple Queue Demonstration

```
21       int          count;
22       QUEUE_NODE* rear;
23      } QUEUE;
24
25  // Function Declarations
26  void insertData (QUEUE* pQueue);
27  void print      (QUEUE* pQueue);
28  bool enqueue    (QUEUE* pList, int  dataIn);
29  bool dequeue    (QUEUE* pList, int* dataOut);
30
31  int main (void)
32  {
33  // Local Declarations
34  QUEUE* pQueue;
35
36  // Statements
37     printf("Beginning Simple Queue Program\n");
38
39     pQueue = malloc(sizeof(QUEUE));
40     if (!pQueue)
41        printf("Error allocating queue"), exit(100);
42
```

PROGRAM 15-16 Simple Queue Demonstration

```
43      pQueue->front = NULL;
44      pQueue->count = 0;
45      pQueue->rear  = NULL;
46
47      insertData (pQueue);
48      print      (pQueue);
49
50      printf("\nEnd Simple Queue Program\n");
51      return 0;
52  }   // main
```

```
Results:
Beginning Simple Queue Program
Creating numbers:  854 763 123 532  82
Queue contained:   854 763 123 532  82
End Simple Queue Program
```

PROGRAM 15-17 Insert Data

```
 1  /* ================== insertData ==================
 2     This program creates random number data and
 3     inserts them into a linked list queue.
 4        Pre  pQueue is a pointer to first node
 5        Post Queue created and filled
 6  */
 7  void insertData (QUEUE* pQueue)
 8  {
 9  // Local Declarations
10     int  numIn;
11     bool success;
12
13  // Statements
14     printf("Creating numbers: ");
15     for (int nodeCount = 0; nodeCount < 5; nodeCount++)
16         {
17          // Generate random number
18          numIn  = rand() % 999;
19          printf("%4d", numIn);
```

PROGRAM 15-17 Insert Data

```
20          success = enqueue(pQueue, numIn);
21          if (!success)
22             {
23              printf("Error 101: Out of Memory\n");
24              exit (101);
25             } // if
26         } // for
27     printf("\n");
28     return;
29  }  // insertData
```

PROGRAM 15-18 Print Queue

```
1  /* ====================== print ====================
2     This function prints a singly linked queue.
3        Pre     pQueue is pointer to valid queue
4        Post    data in queue printed
5  */
6  void print (QUEUE* pQueue)
7  {
8  // Local Declarations
9     int printData;
10
11 // Statements
12    printf("Queue contained:  ");
13    while (dequeue(pQueue, &printData))
14        printf("%4d", printData);
15    return;
16 }  // print
```